Understanding Grand Central Dispatch in Depth

Session 408

These are confidential sessions—please refrain from streaming, blogging, or taking pictures.

Understanding Grand Central Dispatch in Depth

Kevin Van Vechten
Core OS

Dave Zarzycki
Core OS

Understanding GCD in Depth

- Technology overview
- Event sources
- Advanced topics
- Performance and tuning

Related Sessions

Programming with Blocks and Grand Central Dispatch	Marina Wednesday 2:00PM
Migrating Your Application to Grand Central Dispatch	Pacific Heights Thursday 2:00PM

Labs

Grand Central Dispatch Lab	Snow Leopard Lab Thursday 3:30PM

Technology

Overview

Technology Overview

- GCD is part of libSystem.dylib

- Available to all applications

  - #include <dispatch/dispatch.h>

Queues

dispatch_queue_t

dispatch_queue_create
dispatch_queue_get_label
dispatch_get_main_queue
dispatch_get_global_queue
dispatch_get_current_queue
dispatch_main
dispatch_apply
dispatch_apply_f
dispatch_async
dispatch_async_f
dispatch_sync
dispatch_sync_f
dispatch_after
dispatch_after_f

Objects

dispatch_object_t

dispatch_retain
dispatch_release
dispatch_suspend
dispatch_resume
dispatch_debug
dispatch_get_context
dispatch_set_context
dispatch_set_target_queue
dispatch_set_finalizer_f

Groups

dispatch_group_t

dispatch_group_create
dispatch_group_enter
dispatch_group_leave
dispatch_group_wait
dispatch_group_notify
dispatch_group_notify_f
dispatch_group_async
dispatch_group_async_f

Not Objects

Time

dispatch_time_t

dispatch_time
dispatch_walltime

Queues

dispatch_queue_t

dispatch_queue_create
dispatch_queue_get_label
dispatch_get_main_queue
dispatch_get_global_queue
dispatch_get_current_queue
dispatch_main
dispatch_apply
dispatch_apply_f
dispatch_async
dispatch_async_f
dispatch_sync
dispatch_sync_f
dispatch_after
dispatch_after_f

Semaphores

dispatch_semaphore_t

dispatch_semaphore_create
dispatch_semaphore_signal
dispatch_semaphore_wait

Objects

dispatch_object_t

dispatch_retain
dispatch_release
dispatch_suspend
dispatch_resume
dispatch_debug
dispatch_get_context
dispatch_set_context
dispatch_set_target_queue
dispatch_set_finalizer_f

Groups

dispatch_group_t

dispatch_group_create
dispatch_group_enter
dispatch_group_leave
dispatch_group_wait
dispatch_group_notify
dispatch_group_notify_f
dispatch_group_async
dispatch_group_async_f

Sources

dispatch_source_t

dispatch_source_create
dispatch_source_cancel
dispatch_source_testcancel
dispatch_source_merge_data
dispatch_source_get_handle
dispatch_source_get_mask
dispatch_source_get_data
dispatch_source_set_timer
dispatch_source_set_event_handler
dispatch_source_set_event_handler_f
dispatch_source_set_cancel_handler
dispatch_source_set_cancel_handler_f

Not Objects

Time

dispatch_time_t

dispatch_time
dispatch_walltime

Once

dispatch_once_t

dispatch_once
dispatch_once_f

Queues

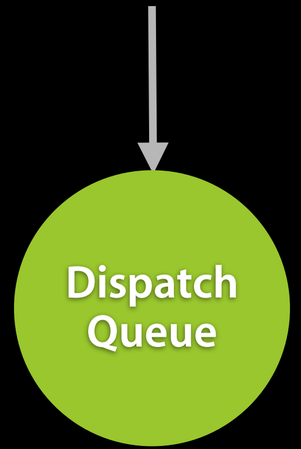
Asynchronous Blocks

Asynchronous Blocks

```
dispatch_async(queue, ^{
    printf(“Hello\n”);
});
```

Your Thread

A = ^{ ... };



Automatic Thread

Simple Asynchronous Blocks

```
dispatch_async(queue, ^{
    myfunc(myctxt);
});
```

Simple Asynchronous Functions

```
dispatch_async_f(queue, myctxt, myfunc);
```

Functions and Contexts

- All GCD API have block and _f variants
- Use context pointer and function pointer instead of block objects
- The block based API are wrappers around the _f procedures

Completion Callbacks

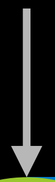
Completion Callbacks

- Continuation of code flow
- Propagate results back to the caller

Main Queue

Main Thread

call = ^{ ... };



Dispatch Queue

callback = ^{ ... };

Automatic Thread

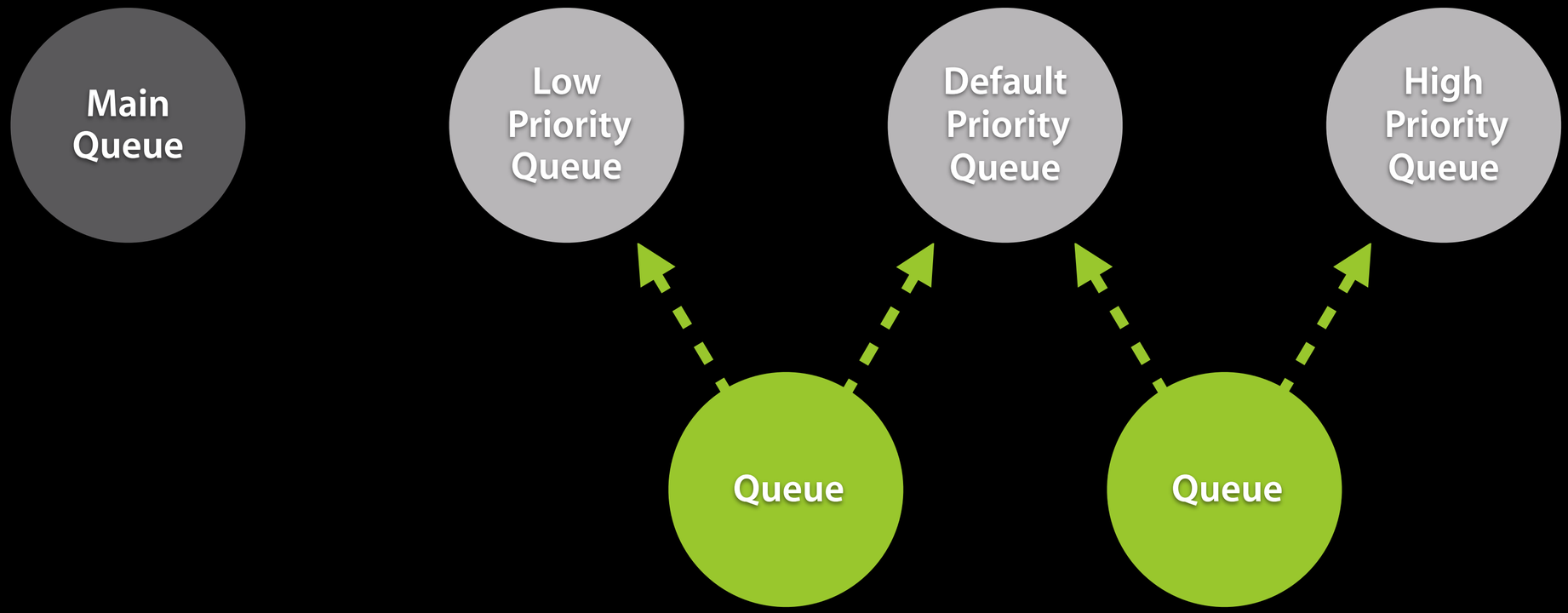
Completion Callbacks

```
reply_q = dispatch_get_main_queue();

dispatch_async(queue, ^{
    result = myfunc(myctxt);
    dispatch_async(reply_q, ^{
        show_result(result);
        free(result);
    });
});
```

Target Queues

Target Queues

- Destination for executing blocks and functions
  - Private queues—default priority global queue
  - Global queues—self
- Global queues map GCD activity to real threads
  - DISPATCH_QUEUE_PRIORITY_HIGH
  - DISPATCH_QUEUE_PRIORITY_DEFAULT
  - DISPATCH_QUEUE_PRIORITY_LOW


Main Queue
Low Priority Queue
Default Priority Queue
High Priority Queue
Queue
Queue

Target Queues

```
dispatch_queue_t queue;
dispatch_queue_t target;
```

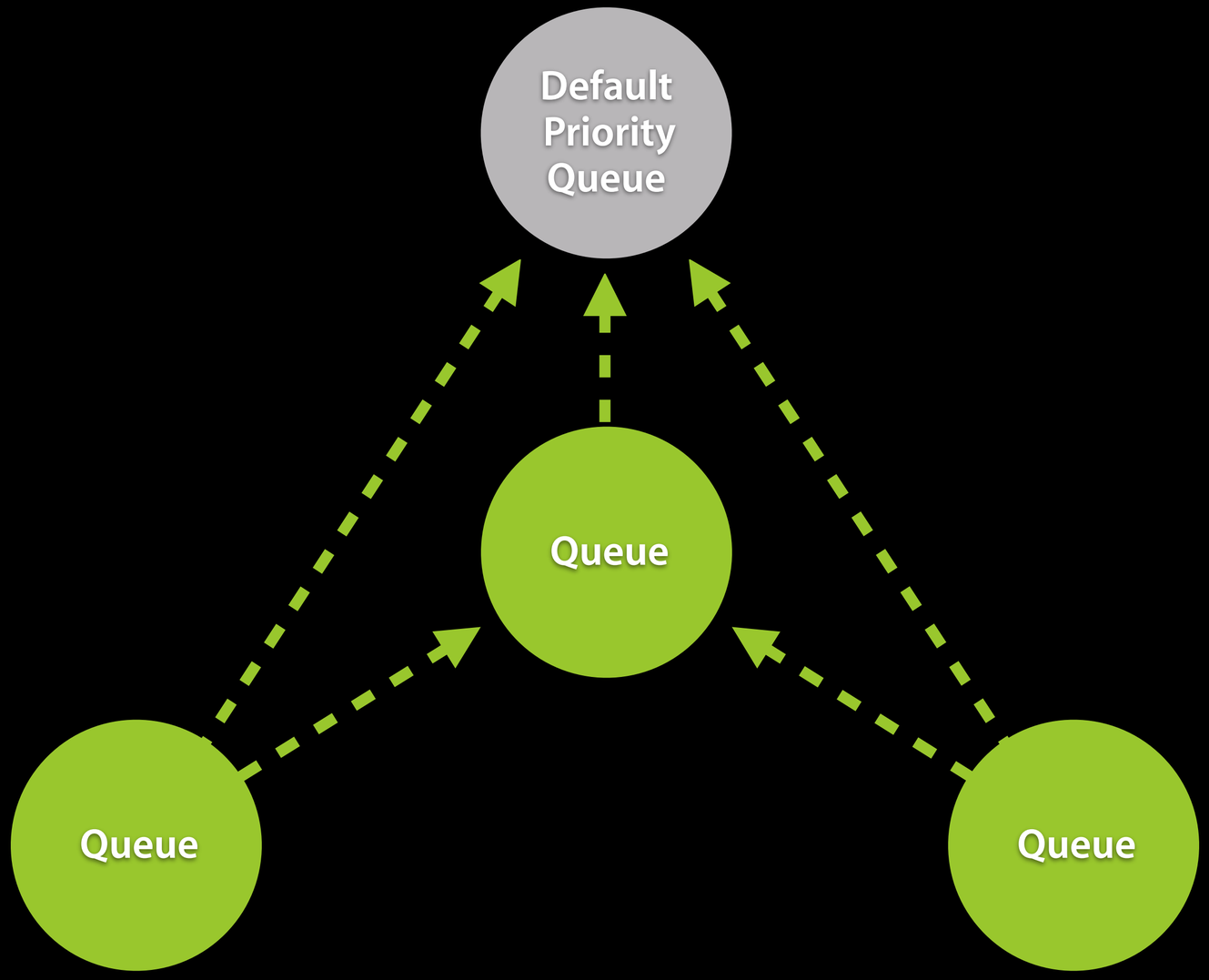
```
queue = dispatch_queue_create(“com.example.test”, NULL);
```

```
target = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_LOW, 0);
```

```
dispatch_set_target_queue(queue, target);
```

Target Queues

- Arbitrary hierarchies are supported
- FIFO scheduling of subordinate queues
- Creating loops is undefined


Default
Priority
Queue
Queue
Queue
Queue

Target Queues

- Why stack serial queues?
- For example:
  - One queue per disk
  - One queue per partition
  - The partition queues should target the disk queue

Suspend and Resume

Suspend and Resume

- Suspend block dequeue *asynchronously*
  - Takes effect between blocks (nonpreemptive)

- Target queue hierarchies ***can*** make the suspension state predictable
  - Serial queues can predictably suspend objects that target them

Suspend and Resume

```
dispatch_suspend(queue);
```

```
dispatch_async(queue, ^{ /* A */ });
```

```
dispatch_resume(queue);
```

Suspend and Resume

```
dispatch_async(queue, ^{ /* A */ });

dispatch_async(queue, ^{ /* B */ });

dispatch_async(queue, ^{ /* C */ });

dispatch_suspend(queue);
```

Nondeterministic whether A, B, or C has run

Objects

Reference Counting

Reference Counting

- Objects are reference counted and freed when count goes to zero
  - `dispatch_retain(object);`
  - `dispatch_release(object);`
- ***Always*** retain and release around asynchronous operations
  - Objects captured by blocks must be retained by your code

Reference Counting

```
dispatch_retain(reply_q);
dispatch_async(queue, ^{
    result = myfunc(myctxt);
    dispatch_async(reply_q, ^{
        show_result(result);
        free(result);
    });
    dispatch_release(reply_q);
});
```

Reference Counting

- GCD retains queue and block parameters, however

- Synchronous calls may assume that the caller holds a reference

```
queue = dispatch_queue_create(“com.example.test”, NULL);
dispatch_sync(queue, ^{
    dispatch_release(queue); // invalid
});
```

Application Contexts

Application Contexts

- Applications may attach custom data to GCD objects
- Optional finalizer callback allows data to be freed with object
  - Called if the context pointer is non-NULL
  - Called on the target queue of the object

Application Contexts

```
struct mystruct *context = calloc(1, sizeof(struct mystruct));

context->x = 50;
context->y = 100;
context->z = 25;
```

```
dispatch_set_context(queue, context);
```

```
dispatch_set_finalizer_f(queue, free);
```

```
dispatch_async(queue, ^{
        struct mystruct *c = dispatch_get_context(queue);
        printf(“x: %d, y: %d, z: %d\n”, c->x, c->y, c->z);
});
```

Semaphores

Semaphores

- Counting semaphore
- Classic synchronization technique
- Initial value determines usage scenario
  - Zero for synchronizing completion
  - Non zero for constraining the width of execution
- ‘Signal’ increments the value
- ‘Wait’ decrements the value

Semaphores

- Signaling
  - Increments the value
  - Returns if the result is greater than zero
  - Otherwise, wake a waiting thread
- Waiting
  - Decrements the value
  - Returns if the result is greater than or equal to zero
  - Otherwise, sleep until a signal occurs

Semaphores

Completion synchronization

```
semaphore = dispatch_semaphore_create(0);
```

```
dispatch_async(queue, ^{
        do_something();
        dispatch_semaphore_signal(semaphore);
});
```

```
// maybe do something else
```

```
dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);

dispatch_release(semaphore);
```

Semaphores

Constraining the width of execution

```
semaphore = dispatch_semaphore_create( getdtablesize() / 2);
```

```
dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
```

```
fd = open(...);
do_something(fd);
close(fd);
```

```
dispatch_semaphore_signal(semaphore);
```

Do not enter critical section until resource is available

Semaphores

- ‘Signal’ increments the value
- ‘Wait’ decrements the value

- *Remember*—Initial value determines usage scenario
  - Zero for synchronizing completion
  - Non zero for constraining the width of execution

Groups

Groups

- Review of the "Intro to Blocks and GCD" presentation 406
  - `dispatch_group_async(group, queue, ^{ ... });`
- Track multiple blocks as a group
  - Empty group notification
- Blocks may run on different queues
- Blocks may add more blocks to the group
  - "Recursive decomposition"

Groups

- Manually managing groups via enter/leave API
  - Does ***not*** implicitly retain/release the group

```
group = dispatch_group_create();

dispatch_group_enter(group);
dispatch_async(queue, ^{
    // my work
    dispatch_group_leave(group);
});
```

Once

Once

- Ensure that multiple threads:
  - Execute a block only once
  - Do not proceed until the block completes
- Usage scenarios:
  - Safely initialize globals at run-time
  - Lazy resource allocation

Once

```
dispatch_queue_t get_my_background_queue(void)
{
    static dispatch_once_t once;
    static dispatch_queue_t my_queue;
    dispatch_once(&once, ^{
        my_queue = dispatch_queue_create(
                “com.example.background”, NULL);
    });
    return my_queue;
}
```

Time

Time

- Special constants
  - DISPATCH_TIME_NOW
  - DISPATCH_TIME_FOREVER
- Opaque dispatch_time_t
  - Nanoseconds relative to now
  - Nanoseconds relative to UNIX epoch (struct timespec)

Time

Creating relative timeouts

```
dispatch_time_t timeout;
```

```
// 30 second timeout from now
timeout = dispatch_time(DISPATCH_TIME_NOW, 30ull * NSEC_PER_SEC);
```

```
// 5 seconds before that
timeout = dispatch_time(timeout, -5ll * NSEC_PER_SEC);
```

```
// forever is forever (delta has no effect)
timeout = dispatch_time(DISPATCH_TIME_FOREVER, -1000 );
assert(timeout == DISPATCH_TIME_FOREVER);
```

Time

Creating absolute timeouts

```
dispatch_time_t timeout;
struct tm tm = {};
struct timespec ts = {};

strptime(“2009-06-11 10:15”, “%Y-%m-%d %H:%M”, &tm);

ts.tv_sec = mktime(&tm);

timeout = dispatch_walltime(&ts, 0);
```

Time

Using timeouts

- Waiting API takes timeout parameter
  - `dispatch_group_wait(group, timeout);`
  - `dispatch_semaphore_wait(semaphore, timeout);`

- Deferred block execution

```
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, 90000), queue, ^{
    printf(“Deferred Hello World by 90µs\n”);
});
```

- Interval timers

Sources

Sources

- Monitor events
  - Internal and external to application
- Deliver callback blocks to target queue
  - Alternately, code can use _f procedural variants
- May be suspended and resumed
- External events based on BSD kqueue events
  - File descriptors, signals, processes, etc.

Sources

Creating an interval timer

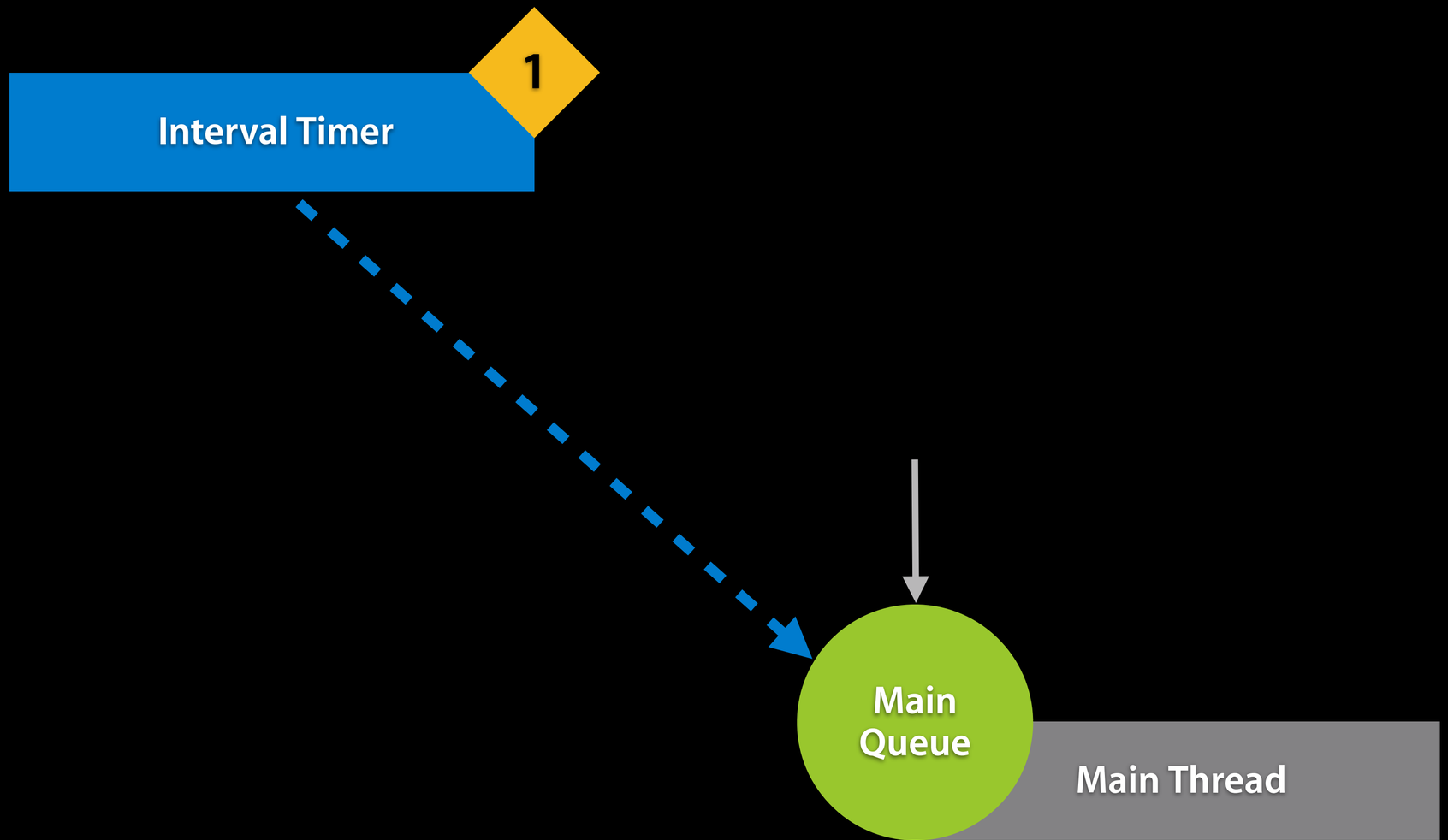
```
dispatch_source_t source;

source = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER,
            0, 0, dispatch_get_main_queue());
```

```
dispatch_source_set_event_handler(source, ^{
    printf(“ping.\n”);
});
```

```
int64_t interval = 30ull * NSEC_PER_SEC;
int64_t leeway = 5ull * NSEC_PER_SEC;
dispatch_time_t start = dispatch_time(DISPATCH_TIME_NOW, interval);
```

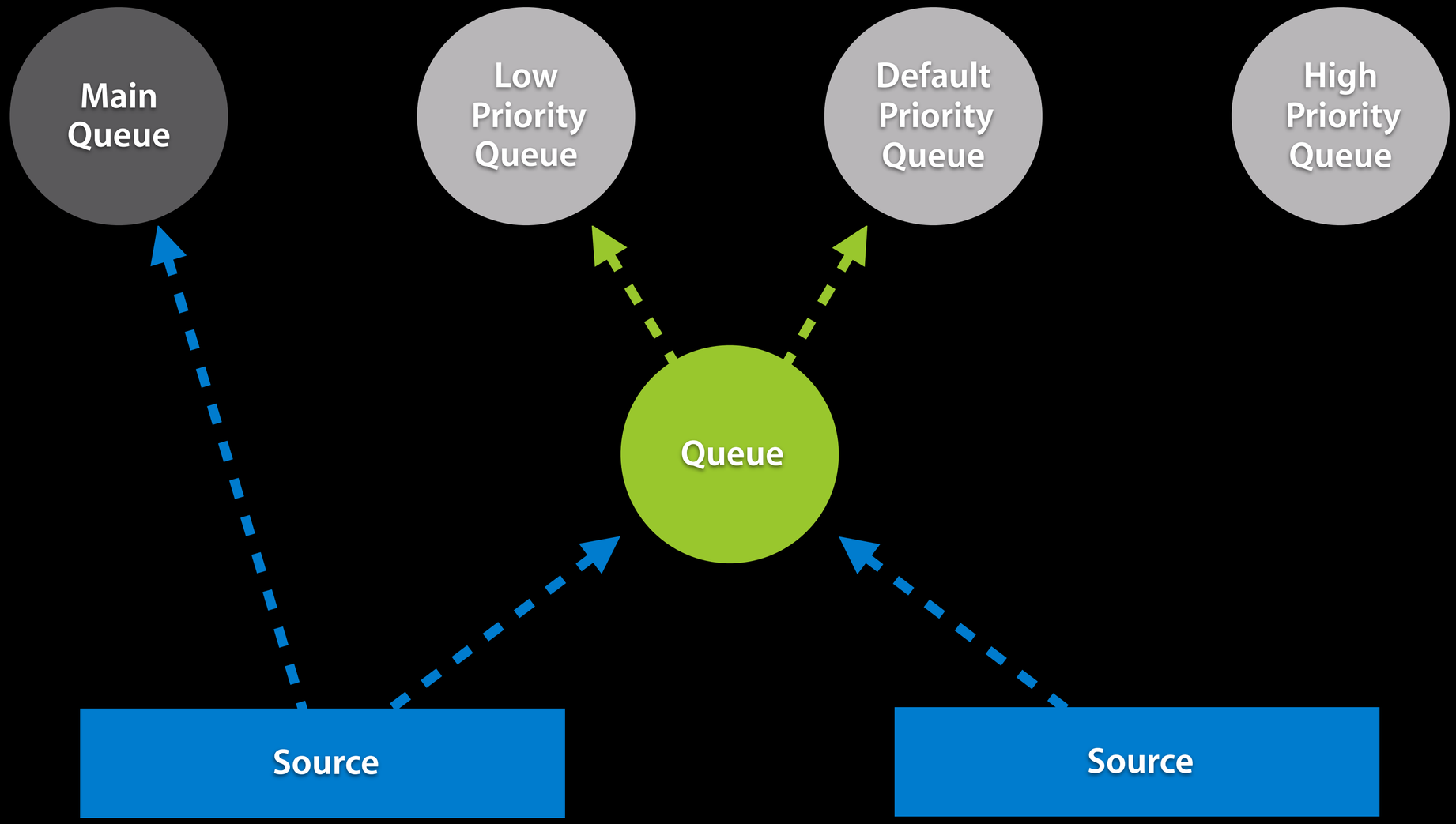
```
dispatch_source_set_timer(source, start, interval, leeway);
```

```
dispatch_resume(source);
```


Interval Timer
1
Main
Queue
Main Thread

Sources

- Target queue passed at creation time
- Changeable:
  - dispatch_set_target_queue()


Main Queue
Low Priority Queue
Default Priority Queue
High Priority Queue
Queue
Source
Source

Sources

Target queue

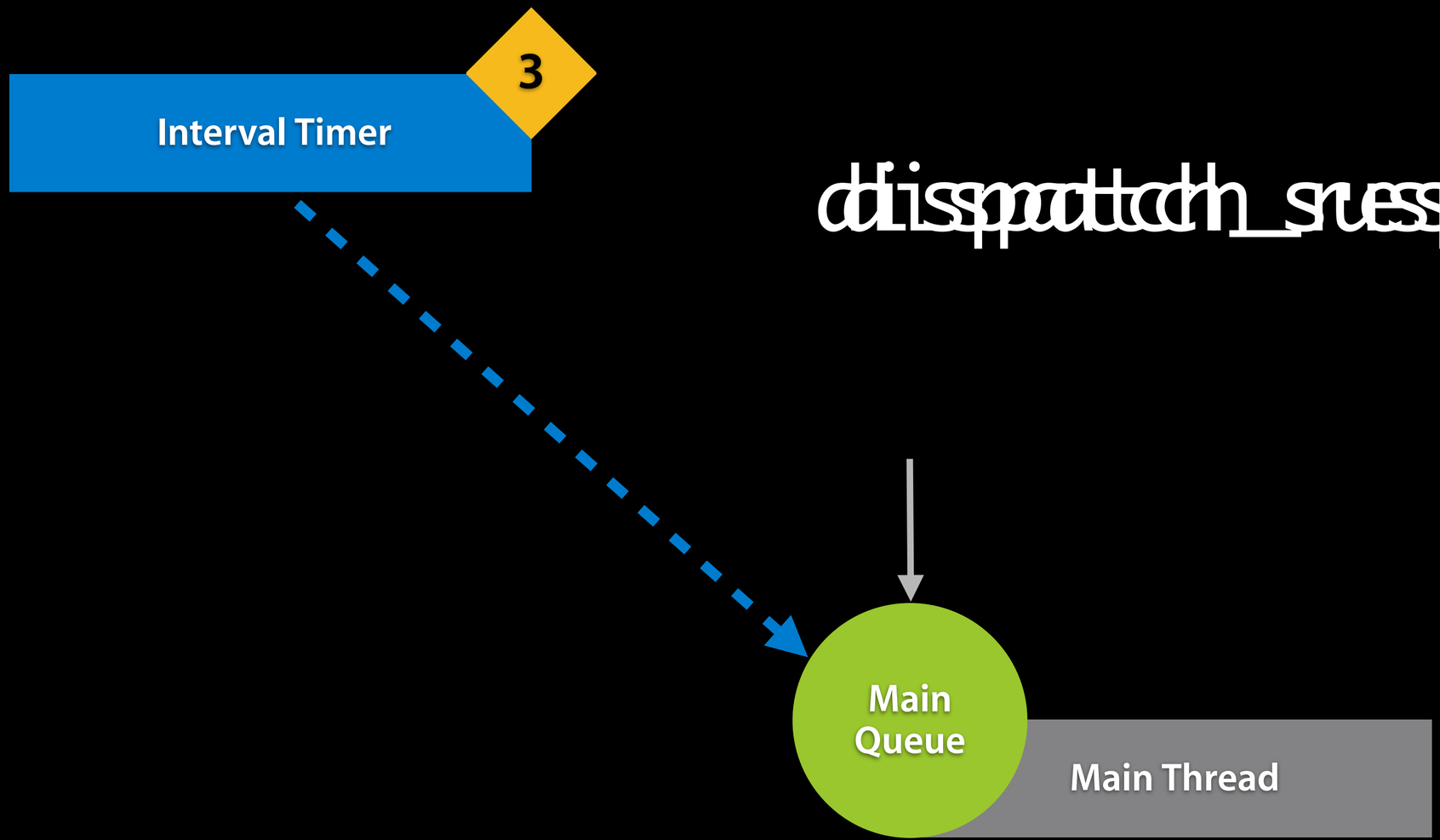
```
dispatch_source_t source;

source = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER,
            0, 0, dispatch_get_main_queue());

dispatch_source_set_event_handler(source, ^{
    printf(“ping.\n”);
});

int64_t interval = 30ull * NSEC_PER_SEC;
int64_t leeway = 5ull * NSEC_PER_SEC;
dispatch_time_t start = dispatch_time(DISPATCH_TIME_NOW, interval);

dispatch_source_set_timer(source, start, interval, leeway);

dispatch_resume(source);
```

Sources

- Event handler is non-reentrant

- Coalesce event data
  - During event handler
  - While suspended

- High performance
  - Data coalesced with hardware atomic operations
  - No ephemeral heap allocations


Interval Timer
3
dispatch_resume()
Main
Queue
Main Thread

Cancellation

Cancellation

- Stops event delivery
  - Does not interrupt event handler (non-preemptive)

- Optional cancellation handler
  - Delivered only once
  - Opportunity to deallocate resources

- Suspension defers cancellation handler

Cancellation

Canceling an interval timer

```
dispatch_source_t source;
__block count = 0;
```

```
source = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER,
            0, 0, dispatch_get_main_queue());
```

```
dispatch_source_set_event_handler(source, ^{
    if (++count <= 5) {
        printf(“ping %d.\n”, count);
    } else {
        dispatch_source_cancel(source);
    }
});
```

```
dispatch_resume(source);
```

Source Types and Event Data

- Internal (application-defined)
- External (monitor system object)

- Behavior depends on source type:
  - Count of number of events
  - Bitmask of event flags

Source Types and Event Data

Constants for internal types

Type	Data
DISPATCH_SOURCE_TYPE_TIMER	count
DISPATCH_SOURCE_TYPE_DATA_ADD	count
DISPATCH_SOURCE_TYPE_DATA_OR	bitmask

Source Types and Event Data

Creating an adding data source

```
progressSource = dispatch_source_create(DISPATCH_SOURCE_TYPE_ADD,
                 0, 0, dispatch_get_main_queue());
```

```
dispatch_source_set_event_handler(progressSource, ^{
    [progressBar incrementBy:dispatch_source_get_data(source)];
});
```

```
dispatch_resume(progressSource);
```

...

```
dispatch_source_merge_data(progressSource, bytesRead);
```

Source Types and Event Data

Constants for external types

Type	Data	Handle
DISPATCH_SOURCE_TYPE_READ	count	int (fd)
DISPATCH_SOURCE_TYPE_WRITE	count	int (fd)
DISPATCH_SOURCE_TYPE_PROC	bitmask	pid_t
DISPATCH_SOURCE_TYPE_SIGNAL	count	int (signo)
DISPATCH_SOURCE_TYPE_VNODE	bitmask	int (fd)
DISPATCH_SOURCE_TYPE_MACH_SEND	bitmask	mach_port_t
DISPATCH_SOURCE_TYPE_MACH_RECV	-	mach_port_t

Source Types and Event Data

Creating a read source

```
FILE *file = fopen(“/usr/share/dict/words”, “r”);
```

```
source = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,
            fileno(file), 0, myqueue);
```

```
dispatch_source_set_event_handler(source, ^{
    myread(file);
    if (feof(file) || ferror(file)) {
        dispatch_source_cancel(source);
    }
});
```

```
dispatch_source_set_cancel_handler(source, ^{
    fclose(file);
});
```

```
dispatch_resume(source);
```

Source Types and Event Data

Creating a read source

- Nonblocking I/O is recommended

- A serial queue is unresponsive while handler is blocked in API
  - See O_NONBLOCK in fcntl(2)
  - See EAGAIN in read(2), write(2), etc.

Advanced Topics

Dave Zarzycki
Core OS

Pipelining

Pipelining

- Multiple queues representing stages in a pipeline
- Shuttle context data between stages
- Potential concurrency across pipeline
- Concurrent tail-calling

Context

File Reader Queue

Default Concurrent Queue

dispatch_apply()

File Writer Queue

Automatic Thread

Pipelining

```
struct mystruct *context = calloc(1, sizeof(struct mystruct));

my_initialize_context(&context);
```

```
dispatch_async_f(reader_queue, context, reader_func);
```

Pipelining

```
void
reader_func(struct mycontext *context)
{
    dispatch_queue_t work_queue;

    myReadInputBuffer(context);

    work_queue = dispatch_get_global_queue(0, 0);

    dispatch_async_f(work_queue, context, work_func);
}
```

Pipelining

```
void
work_func(struct mycontext *context)
{
    dispatch_queue_t queue;

    queue = dispatch_get_global_queue(0, 0);

    dispatch_apply(context->count, queue, ^(size_t idx) {
        parallel_work(idx, context);
    });

    dispatch_async_f(writer_queue, context, writer_func);
}
```

Pipelining

```
void
writer_func(struct mycontext *context)
{
    myWriteOutputBuffer(context);

    // done
}
```

Pipelining

- dispatch_async_f() can be more efficient
- Avoids ephemeral heap allocations
  - No implied Block_copy() and Block_release()

Performance and Tuning

Performance and Tuning

- Memory management
- Imbalanced queues
- Striding
- Benchmarking
- Instruments

Memory Management

Memory Management

- dispatch_async() and dispatch_after()
- 64+ bytes added to the heap per call

```
dispatch_suspend(queue);
for (i = 0; i < (1000 * 1000); i++) {
    dispatch_async(queue, ^{
        printf(“%u\n”, i);
    });
}
// 64+ MB consumed until
// the queue is resumed
```

Memory Management

- Check the size of the DispatchContinuations zone with the “heap” tool
- Just a debugging technique (not API)

```
$ heap 37 | grep 'DispatchContinuations.*Overall'
Zone DispatchContinuations_0x1000ca000: Overall size: 9215KB;
1 nodes malloced for 1KB (0% of capacity); largest unused:
[0x103100000-1023KB]
```

Memory Management

- Track outstanding work
  - Semaphores
  - Groups
  - Callbacks/pipelines
- Pay as you go
- Watch for latency with Instruments
  - Clients must ascertain how much is too much

Imbalanced Queues

Imbalanced Queues

- Queues are a classic computer science paradigm
  - "Producer/consumer"
- Dispatch queues make these designs easy
  - Producer performs dispatch_async() to queue
  - GCD automatically assigns a thread to the consumer
- Optimal performance may require analysis and tuning

Imbalanced Queues

- The GCD automatic thread pool has ramp up and ramp down costs
- Overwhelming the consumer results in memory bloat
- Starving the consumer results in wasted CPU cycles
  - Statistical sampling shows _dispatch_wakeup()

Block Consolidation Can Rebalance Queues

Block Consolidation

```
// Sub-optimal
for (i = 0; i < (1000 * 1000); i++) {
    dispatch_async(queue, ^{
        xyz(i);
    });
}

// Far better
dispatch_async(queue, ^{
    for (i = 0; i < (1000 * 1000); i++) {
        xyz(i);
    }
});
```

Block Consolidation

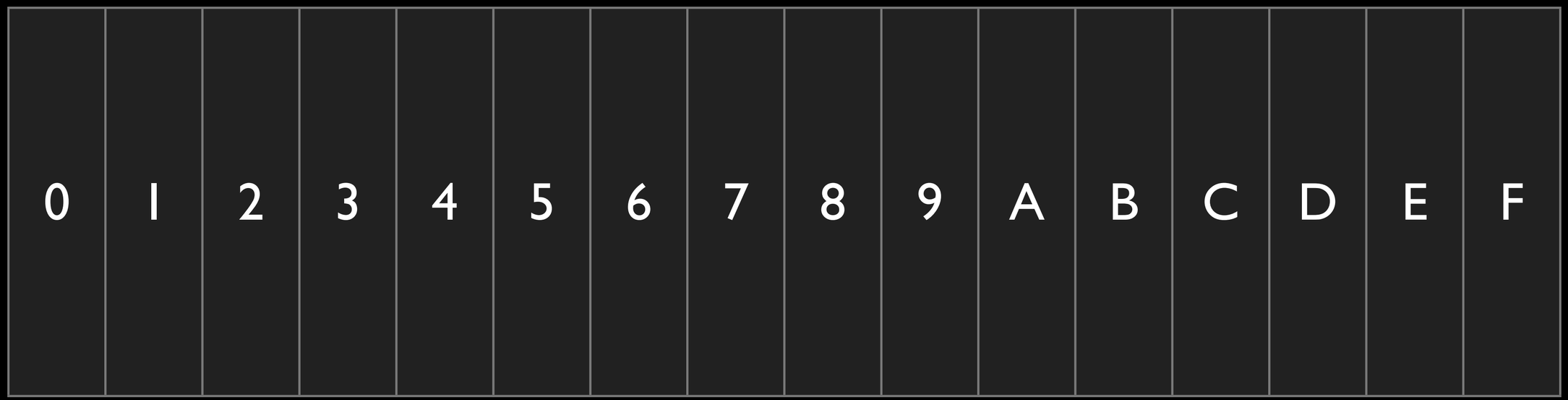
- Cleanup after prototyping or refactoring
- Avoid more heap allocations than necessary

Striding

Striding

- When the ratio of work to scheduling overhead is bad
- *Change the ratio*
- Array specific technique

Striding



Striding

Striding

```
#define STRIDE  137

dispatch_apply(count / STRIDE, queue, ^(size_t idx) {
    size_t j = idx * STRIDE, j_stop = j + STRIDE;
    do {
        printf("%zu\n", j++);
    } while (j < j_stop);
});

for (size_t i = count - (count % STRIDE); i < count; i++) {
    printf("%zu\n", i);
}
```

Benchmarking

Benchmarking

- Good benchmarking is scientific
  - What is being measured?
  - What isn't being measured?
  - What variables aren't easy to control for?
  - Statistical analysis?

Benchmarking

- Start with big and obvious concurrency
- Refine, verify, repeat
  - Always benchmark the serial baseline
  - Performance improvements should justify extra complexity
- Ensure that most of the CPU time is in ***your code***
  - GCD is lightweight, but not free
- Use available tools: Shark, Instruments, sample, etc.

Benchmarking

```
struct timeval start, end, delta;
uint64_t delta_ns;

r = gettimeofday(&start, NULL);
assert(r != -1);
for (i = 0; i < LAPS; i++) {
    test_function();
}
r = gettimeofday(&end, NULL);
assert(r != -1);
timersub(&end, &start, &delta);
delta_ns = (delta.tv_sec * NSEC_PER_SEC
            + delta.tv_usec * NSEC_PER_USEC) / LAPS;
```

Benchmarking

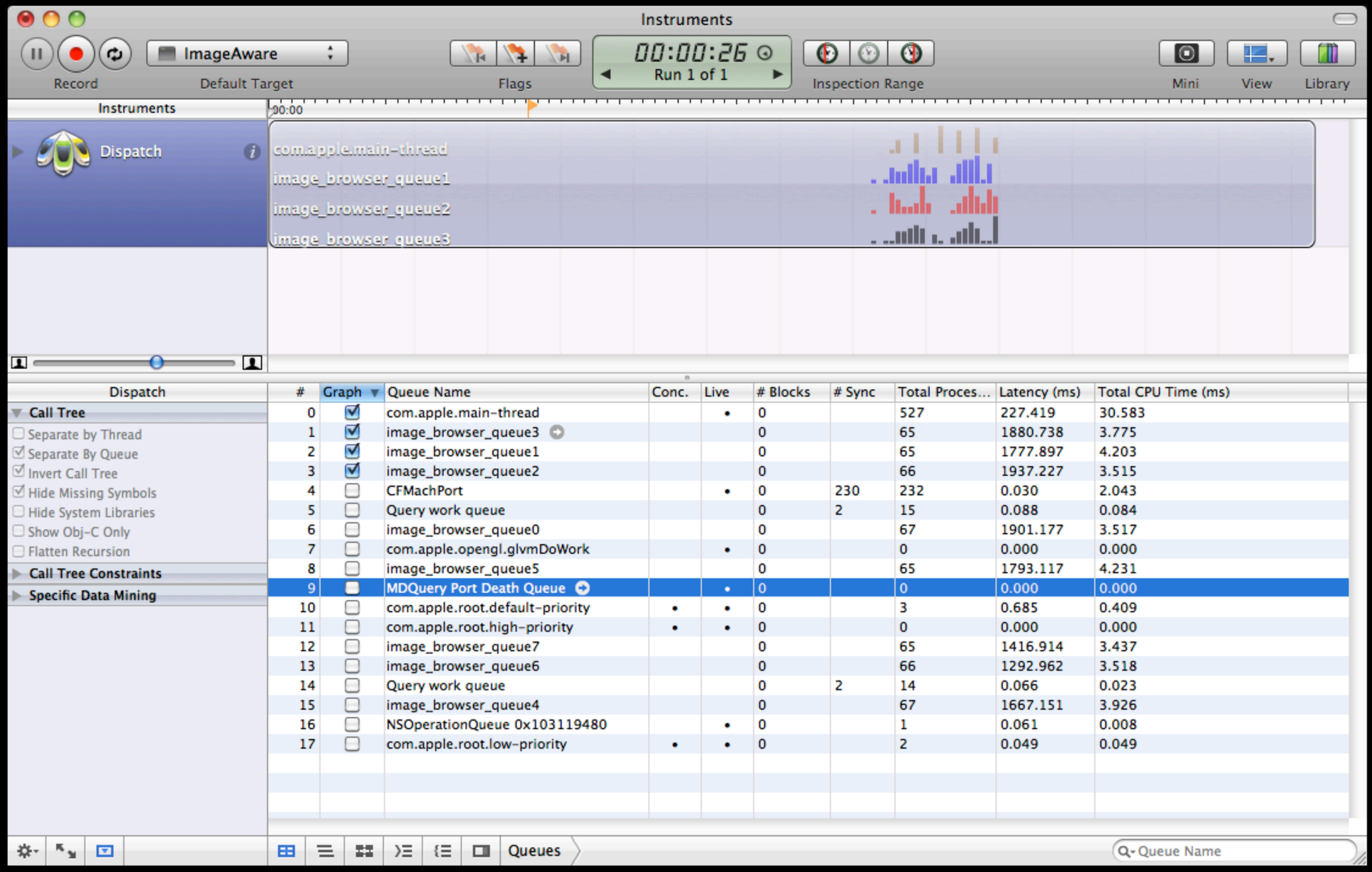
- LAPS should be *big* (one million)
- Why? Long run times average out unusual events
  - Preemption
  - Lazy library initialization
  - Reaching the heap high water mark
  - Other coincidences—both good and bad
- Ensure that the timestamps are *outside* of the loop
  - Mitigate the cost of measuring time

Instruments

Instruments

- Behavior
  - Latency
  - Track which blocks are enqueued on which queues
  - Which blocks are executed synchronously
- Optimization
  - Longest executing blocks
  - Most executing blocks

Instruments



#	Graph	Queue Name	Conc.	Live	# Blocks	# Sync	Total Proces…	Latency (ms)	Total CPU Time (ms)
0	☑	com.apple.main-thread		•	0		527	227.419	30.583
1	☑	image_browser_queue3			0		65	1880.738	3.775
2	☑	image_browser_queue1			0		65	1777.897	4.203
3	☑	image_browser_queue2			0		66	1937.227	3.515
4	☐	CFMachPort		•	0	230	232	0.030	2.043
5	☐	Query work queue			0	2	15	0.088	0.084
6	☐	image_browser_queue0			0		67	1901.177	3.517
7	☐	com.apple.opengl.glvmDoWork		•	0		0	0.000	0.000
8	☐	image_browser_queue5			0		65	1793.117	4.231
9	☐	MDQuery Port Death Queue		•	0		0	0.000	0.000
10	☐	com.apple.root.default-priority	•	•	0		3	0.685	0.409
11	☐	com.apple.root.high-priority	•	•	0		0	0.000	0.000
12	☐	image_browser_queue7			0		65	1416.914	3.437
13	☐	image_browser_queue6			0		66	1292.962	3.518
14	☐	Query work queue			0	2	14	0.066	0.023
15	☐	image_browser_queue4			0		67	1667.151	3.926
16	☐	NSOperationQueue 0x103119480		•	0		1	0.061	0.008
17	☐	com.apple.root.low-priority	•	•	0		2	0.049	0.049

Understanding GCD in Depth

- Technology overview
  - Queue and group review
  - Objects, once, semaphores, groups and time
- Sources
  - Internal and external events
- Performance and tuning

More Information

Michael Jurewitz
Developer Tools and Performance Evangelist
jurewitz@apple.com

Developer Forums
> Mac Development > Grand Central Dispatch
http://devforums.apple.com/community/mac/gcd

Documentation
Concurrency Programming Guide
http://developer.apple.com/

Related Sessions

Understanding Grand Central Dispatch in Depth	Presidio Thursday 9:00AM
Migrating Your Application to Grand Central Dispatch	Pacific Heights Thursday 2:00PM
Designing Your Cocoa Application for Concurrency	Nob Hill Tuesday 3:30PM
What's New in Instruments	Pacific Heights Wednesday 3:30PM
Objective-C and Garbage Collection Advancements	Russian Hill Wednesday 5:00PM
Advanced Debugging and Performance Analysis	Nob Hill Thursday 9:00AM

Labs

Grand Central Dispatch Lab	Snow Leopard Lab Thursday 3:30PM

Q&A

WWDC Session Survey

You can rate your sessions on the WWDC Attendee Site.

Survey forms are located on each session description page at:

developer.apple.com/wwdc/attendee

WWDC09